We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900
Open access books available

185,000
International authors and editors

200M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com



5

Defects in Product Line Models and How to Identify Them

Camille Salinesi and Raúl Mazo
CRI, Panthéon Sorbonne University
France

1. Introduction

The history of software and system development shows that abstraction plays a major role in making complexity manageable (Bosch 2000). Thus, abstracting the common and variable artefacts of an undefined collection of products and organising them into a model may be a good option to manage the complexity of a product line. Product line models improve decision-making processes. In addition, the representation of PLMs in different views improves communication of the actors participating in the product line management (Finkelstein et al. 1992). Nuseibeh et al. (1994) describe views as partial representations of a system and its domain.

Several approaches have been found in literature to represent commonality and variability of a product line. Most of the approaches use features (Kang et al. 1990) as the central concept of product line models. However, other modelling approaches exist like Orthogonal Variability Models (OVM, cf. Pohl et al. 2005), Dopler variability models (Dhungana et al. 2010), Textual Variability Language (TVL, cf. Boucher et al. 2010 and Classen et al. 2010), and constraint-based product line language (Djebbi et al. 2007, Mazo *et al.* 2011e; Salinesi *et al.* 2010b; 2011).

Quality assurance of PLMs has recently been a prominent topic for researchers and practitioners in the context of product lines. Identification and correction of PLMs defects, is vital for efficient management and exploitation of the product line. Defects that are not identified or not corrected will inevitably spread to the products created from the product line, which can drastically diminish the benefits of the product line approach (Von der Maßen and Lichter 2004, Benavides 2007). Besides, product line modeling is an error-prone activity. Indeed, a product line specification represents not one, but an undefined collection of products that may even fulfil contradictory requirements (Lauenroth *et al.* 2010). The aforementioned problems enforce the urgent need of early identification and correction of defects in the context of product lines.

Product line models quality has been an intensive research topic over the last ten years (Von der Maßen & Lichter 2004; Zhang *et al.* 2004; Batory 2005; Czarnecki & Pietroszek 2006; Benavides 2007; Janota & Kiniry 2007; Lauenroth & Pohl 2007; Trinidad *et al.* 2008; Van den Broek & Galvão 2009; Elfaki *et al.* 2009; Kim *et al.* 2011; Liu *et al.* 2011). Usually, to guaranty a certain level of quality of a model, this one must be verified against a collection of criteria

and then, these defects must be corrected. Verifying PLMs entails finding undesirable properties, such as redundancies, anomalies or inconsistencies (Von der Maßen *et al.* 2004). It is widely accepted that manual verification is already tedious and error-prone (Benavides *et al.* 2005). This is even worst when several (often millions) of products are represented altogether in a single specification. Several approaches to automate verification of PLMs have been proposed in order to overcome this limitation. However, despite the relative success of these approaches, there is still a number of pending issues that have motivated the proposal developed in this chapter:

1. Quality assurance techniques from the development of single systems cannot be directly applied to product line specifications because these specifications contain variability. As shows the example presented by Lauenroth *et al.* (2010), a product line may contain requirements R and $\neg R$ at the same time. When a traditional technique is used for verifying this specification, even though those requirements are not included for the same product, a contradiction would be identified since the requirements R and $\neg R$ cannot be fulfilled together. Therefore, it is necessary to take into account the variability of the product line to check whether contradictory requirements can really be part of the same product.
2. The current state of the art on verification is mainly focused on feature models (Kang et al. 1990). Only properties that can be evaluated over feature models represented as boolean expressions are considered in these works. This brushes aside the non-boolean elements of the more sophisticated product line specification formalisms (e.g., integer cardinalities, attributes and complex constraints; cf. Mazo et al. 2011d, Salinesi et al. 2010b, 2011). Current approaches restrict the verification operations to those that can be solved by boolean solvers. The verification is guided by the pre-selected technology and not by the verification requirements themselves. As a result, verification techniques are designed for a limited number of formalisms. These verification techniques are inadequate for many of the existing formalisms, included some used in an industrial context (Djebbi *et al.* 2007; Dhungana *et al.* 2010).
3. Inadequate support for multi-model specification. The size and complexity of industrial product line models motivates the development of this one by heterogeneous teams (Dhungana *et al.* 2006; Segura 2008). Nevertheless, existing tools provide only little support for integrating the models developed by different teams and the subsequent verification of the global model and configurations of products from that model. For instance, a global model that integrates two models must itself have no defects resulting from the integration.

Also in the context of PLs specified with several models, we have identified in our literature review a weak support for verifying the global view of the product line. A product line model has to change over time and in multi-model PLs a change on one of the models can make the global view inconsistent. To the best of our knowledge, existing tools do not provide automated mechanisms for detecting errors on the global PLM as a result of the changes in the different models of the PLM.

This chapter addresses the fourth problem situations aforementioned. To tackle these situations, we present in Section 2 the most relevant concepts used in this chapter, a literature review of related works and the running example to be used in the rest of the chapter. Section 3 presents our typology of verification criteria, which is developed in

Section 4 for the case of single-view product line models, and Section 5 for the case of multi-view product line models. Section 6 presents the evaluation of the approach presented in this chapter.

2. Background and running example

This section presents a literature review on verification of product line models and the corresponding analysis regarding the gaps and challenges identified in each approach. This section also presents a UNIX product line and the corresponding model of the whole or a part of the PL in three different PL modeling languages. The UNIX PL will be used in the rest of this chapter as our running example.

2.1 Verification of product line models

Verifying PLMs entails several aspects. On the one hand, a product line model, independently of the language used to express it, must respect certain properties associated with the domain of product lines. On the other hand, certain properties are associated with the concepts used in the language in which it is expressed. Therefore, some properties of PLMs are independent of the language while other ones are particular to each language. Thus, product line models can be verified from two different points of view. This chapter proposes an approach for PLM verification (Von der Maßen & Lichter 2004; Lauenroth & Pohl 2007; Mendonça *et al.* 2009) in with the engineer selects the verification operations that he/she want to use according to the language in which the model(s) to be verified are specified. In this approach, verification consists in "finding undesirable properties, such as redundant or contradictory information" (Trinidad *et al.* 2008). For instance, PLMs should not be void (i.e., they should allow to configure more than one product) and for the languages with the concept of *optionality*, elements modeled as optional must be really optional (i.e., they should not appear in all the products configured from the PLM).

2.2 Related work

Von der Maßen & Lichter (2004) present an approach to identify redundancies, anomalies and inconsistencies. According to the authors, a feature model contains redundancy, "if at least one semantic information is modeled in a multiple way"; anomalies, "if potential configurations are being lost, though these configurations should be possible"; and inconsistencies, "if the model includes contradictory information". Several cases of redundancies, anomalies and inconsistencies on FMs are identified. In order to validate the approach, the authors use RequiLine, a tool that allows detecting inconsistencies on the domain and on the product configuration level (Von der Maßen & Lichter 2003). The approach was evaluated in "a small local software company" and "in a global player of the automotive industry". However no information about the automating detection of redundancies and anomalies, no details about the sizes of the models or about the technology used to automate the approach or about the results obtained were provided.

Whereas Batory (2005) used grammar and propositional formulas to represent basic FMs and enable truth maintenance systems and SAT solvers to identify contradictory (or inconsistency) predicates to verify that a given combination of features effectively defines a product. In the same line as Batory, Hemakumar (2008) proposed a dynamic solution to find

contradictions, "where errors can be detected during usage and silently reported to model designers". The author proposes an incremental consistency algorithm that incrementally verifies some contradiction properties. The approach consists in verify that a model is contradiction-free if it is k-contradiction free for all k where $0<k\leq n$ (A feature model is k-contradiction free if every selection of k features does not expose a contradiction, for example: "unconditionally" dead features are exposed when $k=1$). When $k=n$, where n is the number of user selectable features, the model has been proven to be contradiction free. However, the incremental consistency algorithm has important practical limits because it is limited to "verify contradiction freedom of models with about 20 or fewer features".

In (Benavides *et al.* 2005a; 2005b; 2006; 2007; Trinidad *et al.* 2008), authors transform FODA models with and without attributes into Boolean expressions. These expressions are executed on Constraint Satisfaction Problem (CSP), Satisfiability (SAT) and Binary Decision Diagrams (BDD) solvers in order to execute analysis and verification operations over feature models. In (Benavides *et al.* 2006) the relationships of the FM are represented as *ifThenElse* constrains on CPS. Despite the originality of this proposal, the constraint representing a feature cardinality (m,n) between the father feature A and its child B (according to their notation: *ifThenElse(A=0;B=0;B in {n,m})*) does not consider that the feature A can itself have a feature cardinality, and in this case the semantic of feature cardinalities is not well represented in the constraint. Authors performed a comparative test between two off–the–shelf CSP Java solvers (JaCoP and Choco). The experiment was executed on five FMs with up to 52 features and in both solvers. The time to get one solution seemed to be linear and the time to get all solutions seemed to be exponential.

Janota & Kiniry (2007) have formalized in higher-order logic (HOL) a "feature model meta-model" that integrates properties of several feature modeling approaches such as attributes and cardinalities. Once the model represented in HOL, author have formulated HOL expressions for root selectivity, existence of a path of selected features from the root to a feature that has been selected, and cardinality satisfaction of a selected feature that each feature model must respect. The approach has been implemented in *Mobius program verification environment*, an Eclipse-based platform for designing, testing, performing various kinds of verification of Java programs and bytecode. Nevertheless, the paper does not provide evidence about the evaluation of the approach, its scalability and its applicability to real cases.

Trinidad *et al.* (2008) mapped FMs into CSP in order to find and diagnose three types of errors: (i) "dead features" are non-selectable features (features that do not appear in any product); (ii) "false optional features", which are features that in spite of being modeled as optional, are always chosen whenever their parents are chosen; and (iii) "void models";a feature model is said to be void if no product can be defined from it. The goal of Trinidad et al. is to detect the above three errors and provide explanations for the cause of these errors. In order to achieve the first goal, authors transform the FM into a CSP expression and then, to query the Choco solver (by means of the FaMa tool) to find the errors. The approach has been evaluated on five FMs up to 86 features. Unfortunately, no details about the scalability and the efficiency of the approach and tool are provided.

Van der Storm (2007) transformed feature diagrams into BDDs in order to check configurations, obtain valid configurations and check consistency of the feature diagram.

Checking the consistency of the feature diagram consists in checking the satisfiability for the BDD logical formulas. Unfortunately, neither details about implementation nor performance nor scalability of the approach are provided in the paper.

Yan et al. (2009) proposed an approach that consists in eliminating verification-irrelevant features and constraints from FMs in order to reduce the problem size of verification, and alleviate the state-space explosion problem. The authors carried out an experiment in which they generated FMs with up to 1900 features. The authors verified the consistency of models and showed that verification is faster when the redundant features had been eliminated. The problem with this approach is that it only considers as redundant, the constraints that contain redundant features, whereas it does not consider typical redundancies such as domain overlapping or cyclic relationships (Salinesi *et al.* 2010; Mazo *et al.* 2011). Besides, (i) the validation of the approach was done with in-house and random build features models, which does not guaranty that the approach works with real world feature models; and (ii) the details about the formalisation and implementation of the approach are not revealed.

Van den Broek & Galvão (2009) analyze FODA product line models using generalized feature trees. In their approach they translate FMs into feature trees plus additional constraints. Once FMs represented in the functional programming language Miranda, they detect the existence of products (void models), dead features and minimal set of conflicting constraints. In FMs with cross-tree constraints, the function to find the number of products belongs to $O(N*2^M)$, where N is the number of features and M is the number of cross-tree constraints. Unfortunatelly, no evaluation of the theoretical calculations of efficiency is reported in the paper. The approach was validated with a feature tree of 13 features and two cross-tree constraints, which is not enough to evaluate the scalability and the usability of the approach on industrial models.

Elfaki *et al.* (2009) propose to use FOL to detect dead features and inconsistencies due to contradictions between include-type and exclude-type relationships in FMs. The innovative point of their work is the suggestion of expressions dealing with both individuals and sets of features.

SPLOT (Mendonca *et al.* 2009b) is a Web-based reasoning and configuration system for feature models supporting group-cardinalities instead of alternative and or-relations. The system maps feature models into propositional logic formulas and uses boolean-based techniques such as BDD and SAT solvers to verify the validity of models (not void) and find dead features.

2.3 Running example

The example taken in this chapter is that of the UNIX operating system, initially presented in (Mazo *et al.* 2011d). UNIX was first developed in the 1960s, and has been under constant development ever since. As other operating systems, it is a suite of programs that makes computers work. In particular, UNIX is a stable, multi-user and multi-tasking system for many different types of computing devices such as servers, desktops, laptops, down to embedded calculators, routers, or even mobile phones. There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, Berkeley (BSD), GNU/Linux, and MacOS X.

The UNIX operating system is made up of three parts: the kernel, the shell and the programs; and two constituent elements: files and processes. Thus, these three parts consist in a collection of files and processes allowing interaction among the parts. The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the file-store and communications in response to system calls. The shell acts as an interface between the user and the kernel, interprets the commands (programs) typed in by users and arranges for them to be carried out. As an illustration of the way the shell, the programs and the kernel work together, suppose a user types *rm myfile* (which has the effect of removing the file *myfile*). The shell searches the file-store for the file containing the program *rm*, and then requests the kernel, through system calls, to execute the program *rm* on *myfile*. The process *rm* removes *myfile* using a specific system-call. When the process *rm myfile* has finished running, the shell gives the user the possibility to execute further commands.

As for any product line, our example emphasizes the common and variable elements of the UNIX family and the constraints among these elements. This example is built from our experience with UNIX operating systems and it does not pretend to be exhaustive, neither on the constituent elements nor on the constraints among these elements. The idea with this PL is, for instance, to look at what utility programs or what kinds of interfaces are available for a particular user. This PL is composed of the following six constraints:

Constraint 1. UNIX can be installed or not and the installation can be from a CDROM, a USB device or from the NET.

Constraint 2. UNIX provides several hundred UTILITY PROGRAMS for each user. The collection of UTILITY PROGRAMS varies even when the UNIX product is full-configured.

Constraint 3. The SHELL is a kind of UTILITY PROGRAM. Different USERS may use different SHELLS. Initially, each USER has a default shell, which can be overridden or changed by users. Some common SHELLS are:

- Bourne shell (SH)
- TC Shell (TCSH)
- Bourne Again Shell (BASH)

For the sake of simplicity will consider only two users in this running example: ROOT_USER and GUEST_USER.

Constraint 4. Some functions accomplished by the UTILITY PROGRAMS are:

- EDITING (mandatory and requires USER INTERFACE)
- FILE MAINTENANCE (mandatory and requires USER INTERFACE)
- PROGRAMMING SUPPORT (optional and requires USER INTERFACE)
- ONLINE INFO (optional and requires USER INTERFACE)

Constraint 5. The USER INTERFACE can be GRAPHICAL and/or TEXTUAL.

Constraint 6. The GRAPHICAL interface is characterized by a WIDTH RESOLUTION and a HEIGHT RESOLUTION that can have the following couples of values [800,600], [1024,768] and [1366,768].

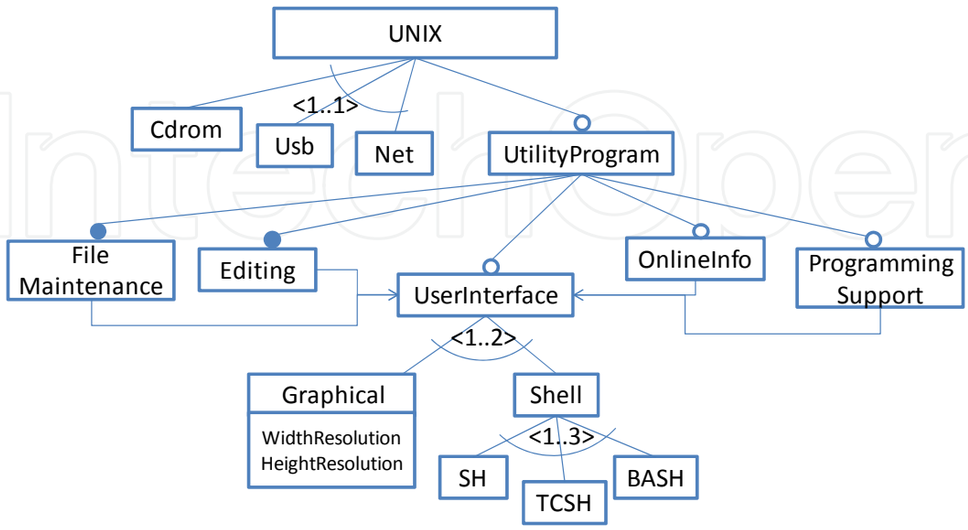
2.3.1 Representation of the UNIX product line as a feature model

Feature Models (FMs) were first introduced in 1990 as a part of the *Feature-Oriented Domain Analysis (FODA)* method (Kang *et al.* 2002) as a means to represent the commonalities and variabilities of PLs. Since then, feature modeling has become a de facto standard adopted by the software product line community and several extensions have been proposed to improve and enrich their expressiveness. A FM defines the valid combinations of features in a PL, and is depicted as a graph-like structure in which nodes represent features, and edges the relationships between them (Kang *et al.* 2002). Two of these extensions are cardinalities (Riebisch *et al.* 2002; Czarnecki *et al.* 2005) and attributes (Streitferdt *et al.* 2003; White *et al.* 2009). Although there is no consensus on a notation to define attributes, most proposals agree that an attribute is a variable with a name, a domain and a value. Attributes are integers, enumerations, and boolean values representing important properties of a feature; as for instance the price, the cost, the width, the height or the time spent to build the corresponding feature. In this chapter we use the group cardinalities grouping bundles of features (cf. *Cdrom*, *Usb* and *Net* in Figure 1). We use the semantic of feature models proposed by (Schobbens *et al.* 2007).

The elements of the FM notation used in this chapter are presented and exemplified as follows:

- *Feature*: A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system (Kang *et al.* 1990). For the sake of simplicity FMs usually comport only the name of the feature; for instance Editing in Figure 1. Every FM must have one root, which is called *root feature* and identifies the product line; for example UNIX in Figure 1.
- *Attribute*: Although there is no consensus on a notation to define attributes, most proposals agree that an attribute is a variable with a name (*Name*), a domain (*Domain*), and a value (consistent with the domain) at a given configuration time. From a technical point of view an attribute must to be differentiated from the other ones by an identifier (*IdAttribute*). For instance in Figure 1, *WidthResolution* and *HeightResolution* are two attributes with a domain determined by the constraint at the bottom of the model.
- *Mandatory:* Given two features *F1* and *F2*, *F1* father of *F2*, a mandatory relationship between *F1* and *F2* means that if the *F1* is selected, then *F2* must be selected too and vice versa. For instance in Figure 1, features *UtilityProgram* and *Editing* are related by a mandatory relationship.
- *Optional:* Given two features *F1* and *F2*, *F1* father of *F2*, an optional relationship between *F1* and *F2* means that if *F1* is selected then *F2* can be selected or not. However, if *F2* is selected, then *F1* must also be selected. For instance in Figure 1, features *UNIX* and *UtilityProgram* are related by an optional relationship.
- *Requires:* Given two features *F1* and *F2*, *F1* requires *F2* means that if *F1* is selected in product, then *F2* has to be selected too. Additionally, it means that *F2* can be selected even when *F1* is not. For instance, *Editing* requires *UserInterface* (cf. Figure 1).
- *Exclusion:* Given two features *F1* and *F2*, *F1* excludes *F2* means that if *F1* is selected then *F2* cannot to be selected in the same product. This relationship is bi-directional: if *F2* is selected, then *F1* cannot to be selected in the same product.
- *Group cardinality:* A group cardinality is an interval denoted <*n..m*>, with *n* as lower bound and *m* as upper bound limiting the number of child features that can be part of a

product when its parent feature is selected. If one of the child features is selected, then the father feature must be selected too. For instance in Figure 1, *Cdrom*, *Usb* and *Net* are related in a <*1..1*> group cardinality.



Graphical→ relation([WidthResolution, HeightResolution], {[800, 600], [1024,768], [1366,768]})

Fig. 1. User model of the UNIX operating system family of our running example

Figure 1 corresponds to the feature representation of the user model of our running example. In this model, a user has the possibility to install a UNIX system using one of the following options: a CD ROM, an USB devise or a network. In addition, users have the possibility to install or not utility programs for file maintenance, edition, online access, and user interface. The user interface may be graphical or command-line (Shell) based; there are three options of command-line interface: SH, TCSH and BASH. The utility programs for user interface, online information and programming support are optional features.

2.3.2 Representation of the UNIX product line as a dopler variability model

The Decision-oriented (Dopler) variability modeling language focuses on product derivation and aims at supporting users configuring products. In Dopler variability models (Dhungana *et al.* 2010a; 2010b), the product line's problem space is defined using *decision models* whereas the solution space is specified using *asset models*. An example of Dopler model is presented in Figure 2. This figure depicts the installation of a UNIX operating system (decision model) and the associated packages (asset model) that can be selected if the UNIX system is installed with a graphical interface. The decision model is composed of four decisions. The first one proposes one of three ways to install a UNIX operating system (with a CD ROM, with a USB or with the Net). The solution of this decision implies the solution of a second decision in which the user must select the utility programs to be installed in the particular UNIX system; in that regard, five utility programs are proposed: one tool for editing, one for file maintenance, one for programming, one for online information access and one shell. If the choice contains the utility program for online information, the user must decide what kind of graphical resolution will be configured and several choices are proposed: *800x600*,

1024x768, *1366x768*. Depending of each selection, the values of the variables corresponding to the width and height resolution will be assigned automatically by means of several decision effects; for instance in Figure 2: *if(GraphicalResolution==800x600) then Width=800*. To finish, the assignation of the width and height resolution must respect a certain number of validity conditions like for instance: *Width* ≥ *800* and *Width* ≤ *1366*. The asset model is composed of seven graphical user interfaces and libraries that can be used in a UNIX graphical interface. The *Tab Window Manager* asset is available for all UNIX implementations with a graphical interface and requires the asset *Motif*; the others assets are optional. The IRIS 4d window manager is based on *Mwm* and *Motif* and therefore requires all of them in order to work in the same way as the *KDE* asset requires the *Qt widget toolkit* to work.

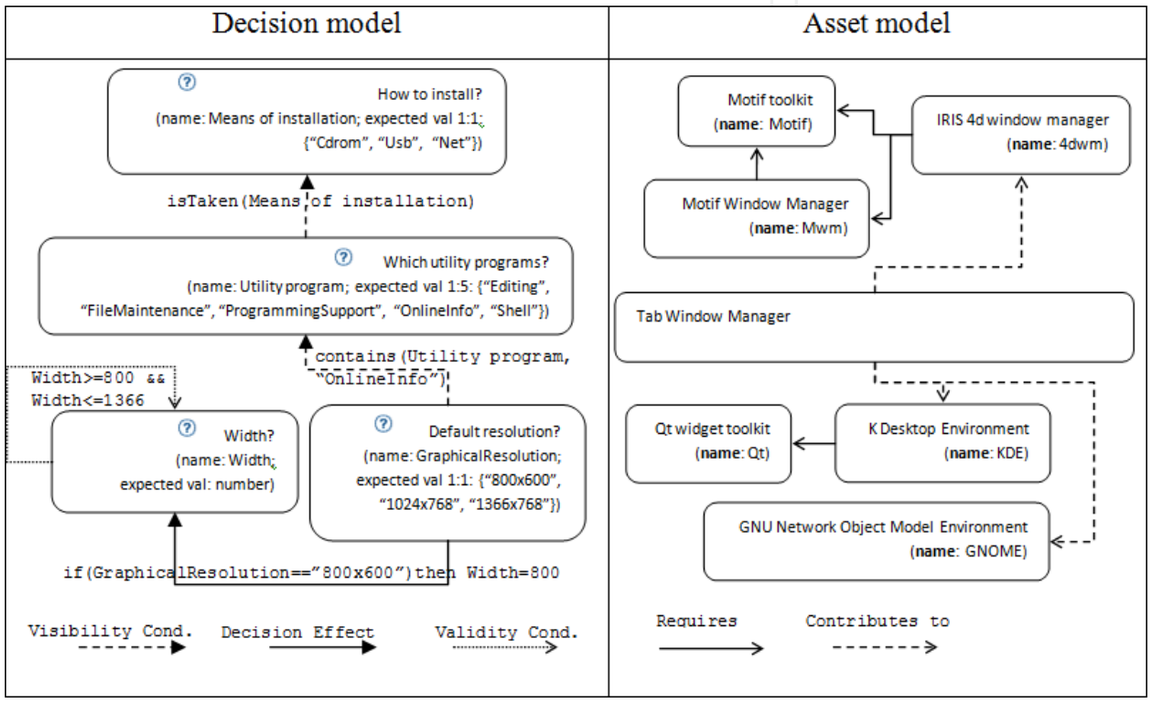


Fig. 2. Example of Dopler Model: Installation of a UNIX System

A **decision model** consists of a set of decisions (e.g., *Which utility programs?* with two attributes: **name** and **expected values**) and dependencies among them (i.e., the **Visibility condition** *isTaken(Means of installation)* forcing the answer of the decision *Utility program* if the decision *Means of installation* is taken). **Assets** allow defining an abstract view of the solution space to the degree of details needed for subsequent product derivation. In a domain-specific metamodel attributes and dependencies can be defined for the different types of assets. Decisions and assets are linked with **inclusion conditions** defining traceability from the solution space to the problem space (e.g., the asset *Tab Window Manager* must be included in the solution space if the option *OnlineInfo* of the decision *Utility program* is selected in a particular configuration). In our integration approach, these inclusion conditions are constraints that will be added to the collection of constraints representing the decision and asset model. Once these constraints are added, both viewpoints of the PL are integrated, and the model is ready to be verified against the typology of verification criteria presented in this chapter.

2.3.3 Representation of the UNIX product line as a constraint program

Constraint Programming (CP) emerged in the 1990's as a successful paradigm to tackle complex combinatorial problems in a declarative manner (Van Hentenryck 1989). CP extends programming languages with the ability to deal with logical variables of different domains (e.g. integers, reals or booleans) and specific declarative relations between these variables called *constraints*. These constraints are solved by specialized algorithms, adapted to their specific domains and therefore much more efficient than generic logic-based engines. A constraint is a logical relationship among several variables, each one taking a value in a given domain of possible values. A constraint thus restricts the possible values that variables can take.

In modern Constraint Programming languages (Diaz & Codognet 2001; Schulte & Stuckey 2008), many different types of constraints exist and are used to represent real-life problems: arithmetic constraints such as $X + Y < Z$, symbolic constraints like *atmost(N,[X1,X2,X3],V)* which means that at most N variables among *[X1,X2,X3]* can take the value V, global constraints like *alldifferent(X1,X2,…,Xn)*meaning that all variables should have different values, and reified constraints that allow the user to reason about the truth-value of a constraint. Solving constraints consists in first reducing the variable domains by propagation techniques that will eliminate inconsistent value within domains and then finding values for each constrained variable in a labeling phase, that is, iteratively grounding variables (fixing a value for a variable) and propagating its effect onto other variable domains (by applying again the same propagation-based techniques). The labeling phase can be improved by using heuristics concerning the order in which variables are considered as well as the order in which values are tried in the variable domains. Consult (Schulte & Stuckey 2008) for more details. Mazo et al. (2011e) present a constraint system to represent product line models by means of abstract constraints where the domain is an argument of the system.

Our running example can also be represented as a constraint program according to the method proposed by Salinesi et al. (2010; 2011) and Mazo et al. (2011d). The resulting model is presented in the following table, where the first column corresponds to each constraint of our example and the second column its representation as a constraint program.

Constraint	CP Representation
C. 1	UNIX ≤ Cdrom + Usb + Net ≤ UNIX
C. 2	UtilityProgram ≤ UNIX
C. 3	Shell = UtilityProgram ∧ Shell ⇒ ((1 * ROOT_USER ≤ ROOT_USERSH + ROOT_USERTCSH + ROOT_USERBASH ≤ 3 * ROOT_USER) ∧ (1 * GUEST_USER ≤ GUEST_USERSH + GUEST_USERTCSH + GUEST_USERBASH ≤ 3 * GUEST_USER))
C. 4	Editing = UtilityProgram ∧ Editing ⇒ UserInterface ∧ FileMaintenance = UtilityProgram ∧ FileMaintenance ⇒ UserInterface ∧ ProgrammingSupport ≤ UtilityProgram ∧ ProgrammingSupport ⇒ UserInterface ∧

Constraint	CP Representation
	OnlineInfo ≤ UtilityProgram ∧ OnlineInfo ⇒ UserInterface ∧ UserInterface ≤ UtilityProgram
C. 5	1 * UserInterface ≤ Graphical + Textual ≤ 2 * UserInterface
C. 6	Graphical = 1 ⇔ (WidthResolution = W1 ∧ HeightResolution = H1) ∧ Graphical = 0 ⇔ (WidthResolution = 0 ∧ HeightResolution = 0) ∧ relation([W1, H1], [[800, 600], [1024, 768], [1366, 768]])

Table 1. UNIX PL represented as a constraint program

3. Typology of verification criteria

Verifying PLMs entails several aspects. On the one hand, a product line model, independently of the language used to express it, must respect certain properties associated with the domain of product lines. On the other hand, certain properties are associated with the fact that each PLM respects the syntactic rules of the language in which it is expressed. Therefore, some properties of PLMs are independent of the language while other ones are particular to each language. In light of this observation, this chapter proposes a typology of PLM verification criteria adapted from the initial version presented in (Salinesi et al. 2010a). The typology presented in Figure 1 is structure in two levels; the top level represents the three categories of verification criteria and the bottom level represents the corresponding operations of the two criteria with more than one operation. This figure indicates that not all PLM verification criteria are equivalent: some are a result of the specification of the PL with a metamodel, whereas others can be used to verify PL specifications independent of the formalism used when they were specified. Besides, some criteria help verifying the ability of PLM to generate all the desired products and only them, whereas others are interested in the quality of PLMs, independently of their semantics (i.e., the collection of all possible products that can be generated from it). This is for example the case with the respect of certain rules providing formality (i.e., absence of ambiguity) at the PLM.

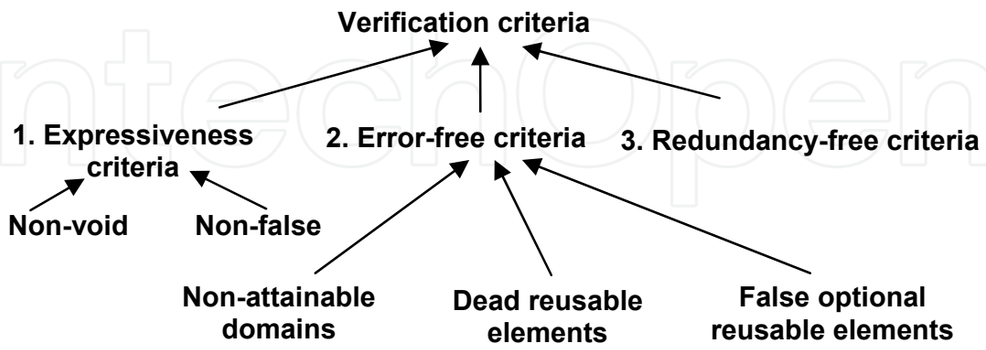


Fig. 3. Typology of verification criteria on PLMs

The outcomes of the typology are multiple:

a. the typology classify the criteria semantic, allowing the identification of similarities and differences among the criteria;
b. the typology makes easier to identify some defects for which no verification criterion is available in the literature. Redundancy of relationships among reusable elements is an example of defect for which no verification criterion has been defined in the literature (at least to our knowledge).
c. the classification behind the typology makes it easier the proposition of a standard and reusable approach to verify the domain-specific criteria of PLMs; and
d. the typology can be used to select the criteria that one wants to use to verify a PLM according to the impact that these criteria have or the expected level of quality of a particular PLM.

The following sections use the typology of verification criteria presented in Figure 3 to develop the verification approach proposed in this chapter.

4. Single-model verification

In order to verify models against the verification criteria identified and calcified in the former section, it is necessary to represent PLMs in a way that is (a) expressiveness-enough to represent the semantics (i.e. the collection of products that can be configured from the PLM) of PLMs, (b) consistent with the formalization of the criteria, and (c) easy to parse with analysis tools. Experience shows that the semantic of every PLM can be represented as a collection of variables over different domains and constrains among these variables. While the variables specify what can vary from a configuration to another one, constraints express under the form of restrictions what combinations of values are allowed in the products.

This section will show how to represent the semantic of PLMs with a constraint based approach, and to verify each and every criterion shown in the typology of the former section on a PLM. The approach will be applied to our feature model example to show how to navigate between the generic specifications of the criteria. The genericity of the approach will be shown by providing examples with other formalisms (cf. Section 5).

Verifying PLMs is about looking for undesirable properties such as redundant or contradictory information. This chapter proposes three domain-specific verification criteria: expressiveness, error-free and redundancy-free. Each domain-specific verification criterion is defined, formalized and exemplified with our running example (cf. Figure 1 and Table 1) as follows.

2.1. Expressiveness: every PLM must allow configuring more than one product, i.e., the model must be not void and the model must be expressive enough to allow configure more than one product (Benavides *et al.* 2005). In case the PLM allows configuring only one product, the PLM, even if it is not considered as a void model, is not expressive enough to be a PLM. Indeed, the purpose of PLMs is to represent at least two products –or there is not reuse. Two verification operations can be used to implement this criterion:

a. **Non-void PLMs**. This operation takes a PLM as input and returns "Void PLM" if the PLM does not define any products. Two alternative techniques have been proposed so far to implement this operation: calculate the number of products (Van den Broek &

Galvão 2009) or ask for a product configuration that meets the constraints of a FM (Benavides *et al.* 2005; Trinidad *et al.* 2008). Our proposal follows along the lines of the latter alternative and is formalized in the following algorithm. It consists in determining if there is at least one product that can be generated by means of a query to an off-the-shelf solver. If the PLM is not void, the solver will return one valid product or *false* otherwise.

```
Non-void _PLM(PLM M, Solver S) {
    S.charge(M);
    Answer = S.getOneSolution();
    If (Answer ≠ "false") {
            Write (Answer);
    }
    Else {
            Write ("Void PLM");
    }
}
```

The execution of this algorithm over the running example gives as result that our UNIX PL is non-void.

b. **Non-false PLMs**. This operation takes a PLM as input and returns "False PLM" if at most one valid product can be configured with it. Although this operation could also help detect when PLMs are void (our precedent operation), the converse is not true. The two operations have then a separate implementation. Our approach consists in asking the solver to generate two products in order to decide if the PLM is false. The algorithm proposed to automate this operation is as follows:

```
Non-false_PLM(PLM M, Solver S) {
    S.charge(M);
    Answer1 = S.getOneSolution();
    If (Answer1 ≠ "false") {
            Answer2 = S.getNextSolution();
            If (Answer2 ≠ "false") {
                    Write (Answer1, Answer2);
            }
            Else {
                    Write ("False PLM");
            }
    }
    Else {
            Write ("False PLM");
    }
}
```

The execution of this algorithm over the running example gives as result that our UNIX PL is a non-false PLM.

2.2. Error-free. The Dictionary of Computing defines an error as "A discrepancy between a computed, observed, or measured value or condition, and the true, specified, or theoretically correct value or condition" (Howe 2010). In PLMs, an error represents a

discrepancy between what the engineer want to represent and the result obtained from the model. For instance, this is the case when the engineer includes a new reusable element (in a given domain) in a PLM, but this element never appears in a product. The error-free criterion can be verified by means of three operations: the first one allows identifying the non-attainable domain values of PLM's reusable elements; the second one allows identifying the dead elements, i.e. elements of the PL that are never used in a product; the third one allows identifying the reusable elements modeled as optional but that appear in all the products of the PL. These operations are presented as follows:

c. **Non-attainable domains**: This operation takes a PLM and a collection of reusable elements as input (all of them by default) and returns the reusable elements that cannot attain one of the values of their domain. Reusable elements can have domains represented as particular values (e.g., 800), intervals of values (e.g., [0..5]), or collections of values (e.g., {0, 800, 1024, 1280}). A non-attainable value of a domain is the value of an element that never appears in any product of the product line. For example, if a reusable element R has the domain [0..1], value 1 is non-attainable if R can never be integrated in a product line it never take the value of 1. Non-attainable values are clearly undesired since they give the user a wrong idea about domain of reusable elements. The approach presented in this chapter can assess the attainability of any reusable elements for all (or parts of) their domain values. This operation was also implemented by Trinidad *et al*. (2008), but only for boolean domains on FMs. Our proposal goes a step further by offering an algorithm for any domain as e.g. needed when using attributes or features whit individual cardinality.
Our algorithm to automate this operation evaluates the domain of each variables of the PLM. For each vale of the domain, the algorithm requests the solver at hand for a solution. If the solver gives a solution for all the values of the variable's domain, the variable is erased from the list of reusable elements with non-attainable domains. Otherwise, the variable, representing a reusable element, is affected with the non-attainable value(s) and kept in the list of reusable elements with non-attainable domains. In each product obtained from the solver, all the variables of the PLM are affected with a particular value of the corresponding domain. Thus, this algorithm takes advantage of that fact and records the answers given by the solver in order to avoid achieving useless requests testing the attainability of domain values that have already been obtained in precedent tests. The corresponding algorithm is as follows:

```
NonAttainableDomains(PLM M, Solver S) {
    S.charge(M);
    For (each variable V ∈ M) {
            For(each Di ∈ domain of V AND not in {PrecedentProducts}){
                    Product = S.getOneSolution("V = Di");
                    If (Product = "false") {
                            Write ("The domain " + Di + " of " + V + " is non-
attainable");
                    }
                    Else {
                            PrecedentProducts += Product;
                    }
```

```
            }
        }
    }
```

For instance in our running example, if when asking for a product with *WidthResolution=800* we get a product

P1 = [UNIX=1, Cdrom=1, Usb=0, Net=0, UtilityProgram=1, FileMaintenance=1, Editing=1, UserInterface=1, Graphical=1, WidthResolution=800, HeightResolution=600, Shell=1, SH=1, TCSH=1, BASH=0, OnlineInfor=0, ProgrammingSupport=0].

This means both that *WidthResolution* can attain the value of 800, and that the rest of variables can attain the values assigned by the solver. Thus, for instance, it is not necessary to ask if the variable *UNIX* can attain the value of 1 or if *HeightResolution* can attain the value of 600.

d. **Dead-free reusable elements**: A reusable element is dead if it cannot appear in any product of the product line. This operation takes as input a PLM and a collection of reusable elements, and it returns the set of dead reusable elements, or false if there is none in the input list. Reusable elements can be dead because: (i) they are excluded by an element that appears in all products (also known as full-mandatory or core reusable elements, c.f. Von der Maßen & Lichter 2004; Benavides *et al.* 2005; Trinidad *et al.* 2008; Van den Broek & Galvão 2009); and (ii) they are wrongly constrained (e.g., an attribute of the feature is > 5 and < 3 at the same time, or a group cardinality is wrong defined). Elfaki et al. (2009) detect dead features by searching only for predefined cases, i.e. defined dead features in the domain-engineering process. Trinidad et al. (2006, 2008) detect dead features by finding all products and then searching for unused features. Van den Broek and Galvão (2009) detect dead features by transforming the FM into a generalized feature tree, and then searching the feature occurrences that cannot be true. To the better of our knowledge there is not details in literature about the way in which the other references have implemented this operation. Our approach evaluates each non-zero value of each reusable element's domain, and reuses each solution obtained from the solver in order to avoid useless computations. If a reusable element cannot attain any of its non-zero values, then the reusable element is dead. The reuse of the solutions previously obtained makes our dead artefacts detection technique scalable as showed below, by contrasts to the state of the art. The corresponding algorithm is presented as follows:

```
DeadReusableElements(PLM M, Solver S) {
    S.charge(M);
    DeadElementsList = all variables of M;
    For (each variable V ∈ DeadElementsList) {
            Product = S.getOneSolution("V > 0");
            If (Product = "false") {
                    Write ("The variable " + V + " is dead");
            }
            Else {
            Erase V and all the other non-zero variables obtained in Product from
DeadElementsList;
            }
    }
}
```

Our algorithm first creates a list of the reusable elements whose dead or non-dead condition is yet to be assessed. For example:
deadElements=[UNIX, Cdrom, Usb, Net, UtilityProgram, FileMaintenance, Editing, UserInterface, Graphical, WidthResolution, HeightResolution, Shell, SH, TCSH, BASH, OnlineInfor, ProgrammingSupport].
Then, our algorithm queries for a configuration based on reusable elements for which we still ignore if they are dead or not, and sieves the selected (and thus alive) elements from this list. For example, to know if *UtilityProgram* is dead or not, it is sufficient to query the solver for a product with *UtilityProgram=1*, which provides a product
P1 = [UNIX=1, Cdrom=1, Usb=0, Net=0, UtilityProgram=1, FileMaintenance=1, Editing=1, UserInterface=1, Graphical=1, WidthResolution=800, HeightResolution=600, Shell=1, SH=1, TCSH=1, BASH=0, OnlineInfor=0, ProgrammingSupport=0].
This means not only that the reusable element *UtilityProgram* is not dead, but also that the other elements with values different from 0 are not dead. Therefore these elements can be sieved from the list of dead elements. The test can be repeated until all elements are sieved. For example querying for products with *Usb =1*, the solver provides another product which means that this reusable element is not dead either. According to our algorithm, the variable *Usb*, and all the other non-zero variables, must be erased from the list of dead elements. At this point the list of dead elements is empty, which means that there are no dead elements in the product line model.
The purpose of the aforementioned list is to reduce the number of queries. For instance in this example, only two queries were necessary to evaluate all reusable elements. In contrast, 17 queries would have been required in the current state of the art algorithm. However, it is not possible to calculate in advance how many queries would be needed, or even, to guaranty that the minimal number of queries will be executed, as this depends on the configuration generated by the solver.

e. **False optional reusable elements**: a reusable element is false optional if it is included in all the products of the product line despite being declared optional (Von der Maßen & Lichter 2004; Benavides *et al.* 2005; Trinidad *et al.* 2008). This operation takes a PLM and a collection of reusable elements modeled as optional as input, and returns the set of false optional reusable elements, or *false* if no one exists. Trinidad et al. (2006, 2009) detect false optional features based on finding all products and then searching for common features among those which are not assigned as common. To verify if an optional reusable element is false optional, we query for a product that does not contain the reusable element at hand (setting the feature's value to 0). If there is no such product, then the reusable element we are evaluating is indeed false optional.

```
FalseOptionalReusableElements(PLM M, Solver S) {
    S.charge(M);
    FalseOptionalElementsList = all optional elements of M;
    For (each variable V ∈ FalseOptionalElementsList) {
            Product = S.getOneSolution("V = 0");
            If (Product = "false") {
                    Write (V + " is false optional");
            }
            Else {
                    Erase V and all the other variables with a Zero affectation into
Product, from DeadElementsList;
```

```
            }
        }
    }
```

For example if we want to know whether the optional reusable component *Usb* is false optional of not, it is sufficient to request for a product without this component (*Support_usb=0*). The solver, in this case, returns the product *P1 = [UNIX=1, Cdrom=1, Usb=0, Net=0, UtilityProgram=1, FileMaintenance=1, Editing=1, UserInterface=1, Graphical=1, WidthResolution=800, HeightResolution=600, Shell=1, SH=1, TCSH=1, BASH=0, OnlineInfor=0, ProgrammingSupport=0]*, which means that this optional reusable element can take the value of 0, it is, be effectively optional.

2.3. Redundancy-free: according to the Oxford dictionary something redundant is something "able to be omitted without loss of meaning or function" (Oxford University 2008). Therefore, redundancy in a PLM is about the presence of reusable elements and variability constraints among them that can be omitted from the PLM without loss of semantic on the PLM. Redundant constraints in FMs are undesired because, although they do not alter the space of solutions, they may consume extra computational effort in derivation and analysis operations (Yan *et al.* 2009), and they are likely to generate inconsistencies when the PL evolves. For the sake of evolution, it is certainly better detect and correct these redundancies. In order to detect them in a PLM this chapter proposes an operation that takes a PLM and a constraint as input and returns *true* if removing the constraint does not change the space of solutions.

Three alternatives can be implemented to check if a relationship is redundant or not. The naïve algorithm consists in calculating all the products of the PLM with the constraint to check; then, remove the constraint; and calculate all the solutions of the new model. If both results are equal (i.e. exact the same products can be configured with and without the constraint), then the constraint is redundant. This approach is computationally very expensive as it requires (a) to compute all configurations twice and (b) to perform an intersection operation between two potentially very large sets (e.g. 10^{21} configurations for the Renault PLM according to Dauron & Astesana (2010)). Not only this algorithm is not scalable, it is typically unfeasible. The second algorithm, proposed by Yan *et al.* (2009) defines a redundant constraint of a PLM as a constraint in which a redundant reusable element takes part. This approach consists in calculating the redundant reusable elements on feature models — features disconnected from the FM — and then the redundant constraint in this approach are those in which the redundant features take part. Though it yields a solution, this algorithm is not sufficiently general: indeed, only these trivial cases of redundancy are considered. The approach proposed in this chapter is based on the fact that if a system is consistent, then the system plus a redundant constraint is consistent too. Therefore, negating the allegedly redundant relation implies contradicting the consistency of the system and thus rendering it inconsistent (Mazo *et al.* 2011a). This approach is more efficient, and thus more scalable, when applied on large models. Our algorithm is in two steps: first, it tries to obtain a solution with the set of constraints. Then, if a solution exists, we negate the constraint we want to check. In the case where no solution is found, the inspected constraint turns out to be redundant. This alternative to find redundant constraints can be formalized as follows:

```
If (at least 1 product can be configured from PLM M under a collection of constraints C
= {C1,...,Ci}) {
    Write (C |= M);
    Let take Cr ∈ C a constraint to be evaluated;
    If (C without Cr |= M AND C ∪ ¬Cr |≠ M) {
            Write (Cr is redundant);
    }
    Else{
            Write (Cr is not redundant);
    }
}
```

For example, to check if the constraint *UNIX* ≥ *UtilityProgram* (cf. Table 1) is redundant or not, it is sufficient to query the solver for a product. Then, if a product is found, the algorithm proceeds to replace the constraint by its negation (*UNIX* < *UtilityProgram*) and ask again for a product. If the solver does not give a solution (as is the case for our running example), one can infer that the constraint (*UNIX* ≥ *UtilityProgram*) is not redundant.

5. Multi-model verification

Multi-model modeling allows tackling various models and aspects of a system, in particular in the presence of stakeholders with multiple viewpoints (executives, developers, distributors, marketing, architects, testers, etc.; cf. Nuseibeh *et al.* 1994). For example, a UNIX product line can be composed of several models, each one developed by a different team or developing a particular view of the PL. Thus, while the team responsible of the kernel develops a model, the team responsible of the user interface develops another model. Motivated by the fact that (a) this practice is current in industry (Dhungana *et al.* 2010); (b) even if each individual model is consistent, once the models are integrated, they can easily be inconsistent; and (c) the lacks in current state of the art in multi-model PL verification, this chapter proposes a method to verify multi-model PLs. This method is composed of fourth steps: (i) the base models' semantic should be transformed into constraint programs; (ii) once these base models transformed into CP, they may be integrate using the integration strategies and rules appropriates for each language (cf. Mazo et al. 2011a for further details about integration of Dopler models, and Mazo *et al.* 2011d for further details about integration of constraint-based PLMs; and (iii) once the base models integrated, the collection of verification criteria, proposed in Section 4 for single models, can be applied on the integrated model in the same manner as for single models.

The application of these verification criteria over the Dopler model depicted in Figure 2 and the explanation regarding the minor variants are presented as follows:

1. **Non-void model**. This model is not a void because it allows configure at least one product; for instance *C1 = {USB, Editing, ProgrammingSupport, Shell}*
2. **Non-false model**. This model is not a false because it allows configure more than two products; for instance: *C2 = {Cdrom, Editing, OnlineInfo, Shell, Twm, KDE, Qt, GraphicalResolution = "800x600", Width = 800}* and *C3 = {USB, Editing}*.
3. **Non-attainable validity conditions' and domains' values.** This operation either (i) takes a collection of decisions as input and returns the decisions that cannot attain

one or more values of its validity condition; or (ii) takes a collection of assets as input and returns the assets that cannot attain one of the values of its domain. A non-attainable value of a validity condition or a domain is a value that can never be taken by a decision or an asset in a valid product. Non-attainable values are undesired because they give the user a wrong idea of the values that decisions and assets modeled in the product line model can take. In our example of Figure 2, the validity condition *Width ≥ 800 && Width ≤ 1366* determines a very large range of values that can take the variable Width, however this variable can really take three values: 800, 1024 and 1366 which means that values like 801, 802,..., 1023, 1025, ..., 1365 are not attainable values.

4. **Dead reusable elements.** In Dopler language, the reusable elements are Decisions and Assets. This operation takes a collection of decisions and assets as input and returns the set of dead decisions and assets (if some exist) or *false* otherwise. A decision is dead if it never becomes available for answering it. An asset is dead if it cannot appear in any of the products of the product line. The presence of dead decisions and assets in product line models indicates modeling errors and intended but unreachable options. A decision can become dead (i) if its visibility condition can never evaluate to true (e.g., if contradicting decisions are referenced in a condition); (ii) a decision value violates its own visibility condition (e.g., when setting the decision to true will in turn make the decision invisible); or (iii) its visibility condition is constrained in a wrong way (e.g., a decision value is > 5 && < 3 at the same time). An asset can become dead (i) if its inclusion depends on dead decisions, or (ii) if its inclusion condition is false and it is not included by other assets (due to *requires* dependencies to it). Dead variables in CP are variables than can never take a valid value (defined by the domain of the variable) in the solution space. Thus, our approach consists in evaluating each non-zero value of each variable's domain. If a variable cannot attain any of its non-zero values, the variable is considered dead. For instance, in the Dopler model of Figure 2, there are not dead decisions or assets.
5. **Redundancy-free**. In the asset model (cf. the right side of Figure 2) the asset *4dwn* requires *MwM*, which at the same time requires the asset *Motif*, therefore the dependency *4dwm* requires *Motif* is redundant according to the redundancy-free algorithm presented in Section 4.

It is worth noting that the domain-specific operation "false optional-free reusable elements" is not applicable in Dopler models due to the fact that this language does not have explicitly the concept of optional. Decisions and assets are optional in Dopler models according to the evaluation of the visibility conditions (in the case of decisions) and inter-assets dependencies in the case of assets

6. Validation

We performed a series of experiments to evaluate the verification approach proposed in this chapter. The goal was to measure the effectiveness or precision of the defect's detection, the computational scalability and the usability of the approach to verify different kinds of product line models. These measurements are presented in the next sections, grouped by the kind of product line models used to evaluate our approach.

6.1 Single-view models

We assessed the feasibility, precision and scalability of our approach with 46 models, out of which 44 were taken from the SPLOT repository (Mendonca *et al.* 2009b) and the other two models are the Vehicle movement control system (Salinesi *et al.* 2010b) and the Stago model (Salinesi et al. 2011). The sizes of the models are distributed as follows: 32 models of sizes from 9 to 49 features, 4 from 50 to 99, 5 from 100 to 999 and 6 from 1000 to 2000 features. The six largest feature models that we have were not considered in this experiment due to the fact that the solver used does not accept more that 5000 variables. Note that SPLOT models do not have attributes, on the contrary to our two industrial models. Therefore artificial attributes were introduced in a random way, in order to have models with 30%, 60% or 100% of their features with attributes. In order to do that, we created a simple tool[1] that translates models from SPLOT format to constraint programs, and we integrate next the artificial attributes. In order to test that the transformation respects the semantic of each feature model, we compared the results of our models without attributes with the results obtained with the tools SPLOT (Mendonca *et al.* 2009b) and FaMa (Trinidad *et al.* 2008b). In both comparisons we obtained the same results in all the shared functions: detection of void models, dead features, and false optional features. These results show that our transformation algorithm respects the semantic of initial models.

6.1.2 Precision of the detection

Not only must the transformation of FMs into CPs be correct but also the detection of defects. As aforementioned, we compared the results obtained with our tool VariaMos against these obtained with two other tools: SPLOT and FaMa. These comparisons were made over models without attributes due to the fact that original models taken from SPLOT, and also available for FaMa, do not have attributes. In these comparisons we find the same results, for the common verification functions on the three tools, but due to the fact that our own models contain attributes and group cardinalities <*m..n*>, for any *m* and *n* bellowing to non negative integer numbers, a manual inspection were necessary. A manual inspection on two samples of 28 and 56 features showed that our approach identify the 100% of the anomalies with 0% false positive.

6.1.3 Computational scalability

The execution time of the verification operations in our tool shows that the performance obtained with our approach is acceptable in realistic situations; because in the worst case, users can execute any verification operation less than 19 seconds for models up to 2000 features. Figure 4 shows the execution time of each one of the six verification operations in the 50 models. In Figure 4 each plot corresponds to a verification operation: Figure 4(1) corresponds to operation 1, Figure 4(2) corresponds to operation 2 and so on. Times in the Y axis are expressed in milliseconds (ms) and X axis corresponds to the number of features. It is worth noting that most of the results overlap the other ones; we avoid the use of a logarithmic scale in the X axis, to keep the real behaviour of the results.

[1] parserSPLOTmodelsToCP.rar available at: https://sites.google.com/site/raulmazo/

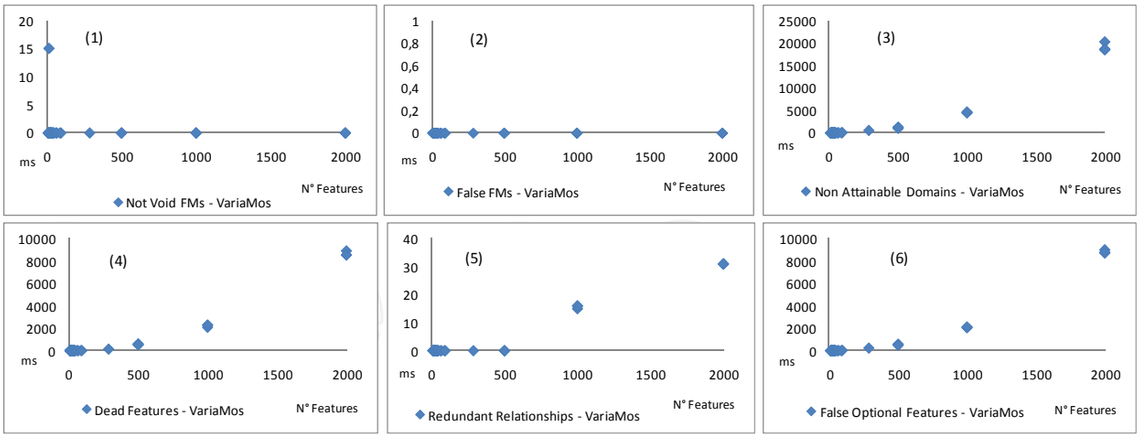


Fig. 4. Execution time of the six verification operations, per number of features

Let us now present the results in more detail. For the models with sizes between 9 and 100 features our approach verified all operations in less than 1 second on average. For the models with sizes between 101 and 500 features verified dead features and false optional features in 0,4 seconds, 1 second to calculate the non attainable domains and 0 milliseconds in the rest of verification operations. It is worth noting ant our solver does not provide time measures of microseconds (10^{-6} seconds); thus, 0 milliseconds (10^{-3} seconds) must be interpreted as less than 1 millisecond. In general, over the 46 FMs, the execution time to detect dead features, false optional features and non attainable domains is inferior than 8,68, 8,82 and 19,09 seconds respectively. For the rest of verification operations, the execution time is inferior to 0,02 seconds even for the largest models. Following the projection of our results, our approach is able to be used in larger FMs with a quadratic increase, in the worst of cases, of the time to execute any verification operation proposed in this paper. To finish, the verification operations like redundant relationships, false feature models and void feature models are executed in less than 0,03 seconds. According to the results of our experiment, we can conclude that our verification approach presented in this chapter is scalable to large FMs.

3.6 The case multi-view models

We also tested our verification approach with two Dopler variability models (Mazo *et al.* 2011a). In both models, we seeded 33 defects in the DOPLER model and 22 defects in the camera model. The defects cover different types of problems to show the feasibility of the verification approach. For instance, the decision *Wizard_height* cannot take the values 1200, 1050, 1024 and 768 and the asset *VAI_Configuration_DOPLER* cannot take the value 1 (is never included for any product), even if these values take part in the corresponding variables' domain. Furthermore, we measured the execution time of applying the approach for both models for the different verification operations as presented below.

Applying our verification approach to the DOPLER model has shown that the model is not void and can generate 23016416 products. However, we discovered 18 defects related with non-attainable domain values and 15 dead decisions and assets (these together are the 33 defects we have seeded before). By applying our verification approach on the digital camera

model we obtained that the model is not void and can generate 442368 products. In this model, we discovered 11 defects related with non-attainable domain values as well as 11 dead decisions and assets (these together are the 22 defects we have seeded before). It is noteworthy that the same number of defects was identified in a manual verification of both models. The automated verification found all of the seeded defects in the DOPLER model and all of the seeded defects in the camera model.

Table 2 shows the number of defects found and the execution time (in milliseconds) corresponding to the verification operations on the models. No defects were found regarding the "Void model", "False model" and "Redundant relationships" operations and the execution time was less than 1 millisecond for each one of these operations in each model. The model transformations from Dopler models to constraint programs took about 1 second for each model.

		Void model	False model	Non-attainable domains	Dead Decisions and Assets	Redundant relationships
DOPLER 81 Variables	Defects	No	No	18	15	No
	Time	0	0	125	47	0
Camera 39 Variables	Defects	No	No	11	11	No
	Time	0	0	16	15	0

Table 2. Results of model verifications: Execution time (in milliseconds) and number of defects found with each verification operation.

In the same way as for the single-view models, the results obtained on multi-view models allow concluding that the verification approach presented in this chapter is scalable to medium Dopler models and give promising expectations on large Dopler models.

5. References

Batory D. (2005). Feature Models, Grammars, and Propositional Formulas. *In Proceedings of the International Software Product Line Conference (SPLC)*, pages 7-20. Rennes, France.

Benavides D. On the Automated Analysis of Software Product Lines Using Feature Models. A Framework for Developing Automated Tool Support. (2007). University of Seville, Spain, PhD Thesis.

Benavides, D., Segura, S., Trinidad, P., and Ruiz-Cortés, A. (2006). Using Java CSP solvers in the automated analyses of feature models. In Post-Proceedings of The Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE). LNCS 4143.

Benavides, D., Segura, S., Ruiz-Cortés, A. (2010). Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems journal, Volume 35 , Issue 6, Elsevier, PP. 615-636

Benavides, D.; Trinidad, P. & Ruiz-Cortés, A. (2005). Automated Reasoning on Feature Models. *In Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005*. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg.

Benavides, D.; Ruiz-Cortés, A.; Trinidad, P. (2005). Using constraint programming to reason on feature models. In The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE 2005, pages 677–682.

Bosch, J. (2000). *Design and Use of Software Architectures. Adopting and evolving a product-line approach*. Addison-Wesley.

Cabot, J. & Teniente, E. (2006). Incremental evaluation of ocl constraints. *In Dubois, E., Pohl, K. (eds.) CAiSE'06*. LNCS, vol. 4001, pp. 81–95. Springer, Heidelberg.

Clements, P. & Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison Wesley, Reading, MA, USA.

Czarnecki, K.; Pietroszek, K. (2006). Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints, 5th Int. Conference on Generative Programming and Component Engineering.

Czarnecki, K.; Helsen, S. & Eisenecker, U. W. (2005). Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1), pp. 7-29.

Dauron, A. & Astesana, J-M. (2010). Spécification et configuration de la ligne de produits véhicule de Renault. *Journée Lignes de Produits*. Université Pantéon Sorbonne, France.

Dhungana, D.; Grünbacher, P. & Rabiser R. (2010). The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study. *Automated Software Engineering* (in press; doi: 10.1007/s10515-010-0076-6).

Dhungana, D.; Heymans, P. & Rabiser, R. (2010). A Formal Semantics for Decision-oriented Variability Modeling with DOPLER. *Proc. of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Linz, Austria, ICB-Research Report No. 37, University of Duisburg Essen, 2010, pp. 29-35.

Dhungana, D., Rabiser, R. & Grünbacher, P. (2006). Coordinating Multi-Team Variability Modeling in Product Line Engineering. *In 2nd International Workshop on Supporting Knowledge Collaboration in Software Development (KCSD)*, Tokyo, Japan.

Diaz, D. & Codognet, P. (2001). Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming (JFLP)*, Vol. 2001, No. 6.

Djebbi, O.; Salinesi, C. & Fanmuy, G. (2007). Industry Survey of Product Lines Management Tools: Requirements, Qualities and Open Issues. *Proc. of the International Conference on Requirement Engineering (RE)*, IEEE Computer Society, New Delhi, India.

Djebbi, O. & Salinesi C. (2007). RED-PL, a Method for Deriving Product Requirements from a Product Line Requirements Model. *Proc. of the International Conference CAISE'07*. Norway.

Egyed, A. (2006). Instant consistency checking for UML. *In: International Conf. Software Engineering (ICSE'06)*, pp. 381–390. ACM Press, New York.

Elfaki, A.; Phon-Amnuaisuk, S. & Kuan Ho C. (2009). Using First Order Logic to Validate Feature Model. *Third International Workshop on Variability Modelling of Software-intensive Systems VaMoS*. ICB-Research Report No. 29, Universität Duisburg-Essen, pp. 169-172. Spain.

Finkelstein, A.C.W.; Gabbay, D.; Hunter, A.; Kramer, J. & Nuseibeh, B. (1994) Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, pages 569–578.

Finkelstein, A.; Kramer, J.; Nuseibeh, B.; Finkelstein, L.; Goedicke, M. (1992). Viewpoints: A framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering 2(1).

Griss, M.; Favaro, J. & d'Alessandro, M. (1998). Integrating feature modeling with the RSEB. *In Proceedings of the Fifth International Conference on Software Reuse*. Vancouver, BC, Canada.

Hemakumar, A. (2008). Finding Contradictions in Feature Models. Workshop on the Analysis of Software Product Lines (ASPL).

Howe, D. (2010). *The Free On-line Dictionary of Computing*, 01.06.2011, Available from http://foldoc.org

Janota, M.; Kiniry, J. (2007). Reasoning about Feature Models in Higher-Order Logic, in 11th Int. Software Product Line Conference (SPLC07).

Kang, K.; Cohen, S.; Hess, J.; Novak, W. & Peterson, S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Technical Report CMU/SEI-90-TR-21, Software Engineering Institute*, Carnegie Mellon University, USA.

Kang, K.; Lee, J.; Donohoe, P. (2002). Feature-oriented product line engineering. Software, IEEE, 19(4).

Kim, C.H.P.; Batory, D.; Khurshid, S. (2011). Reducing Combinatorics in Testing Product Lines. Aspect Oriented Software Development (AOSD).

Lauenroth, K.; Metzger, A.; Pohl, K. (2010). Quality Assurance in the Presence of Variability. S. Nurcan et al. (eds.), Intentional Perspectives on Information Systems Engineering, Springer-Verlag, Berlin Heidelberg.

Lauenroth, K. & Pohl, K. (2007). Towards automated consistency checks of product line requirements specifications. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering ASE'07*, USA.

Liu, J.; Basu, S.; Lutz, R. R. (2011). Compositional model checking of software product lines using variation point obligations. Journal Automated Software Engineering, Volume 18 Issue 1.

Matthias, R.; Kai, B.; Detlef, S. & Ilka, P. (2002). Extending feature diagrams with UML multiplicities. *Proceedings of the Sixth Conference on Integrated Design and Process Technology*. Pasadena, CA.

Mazo, R.; Grünbacher, P.; Heider, W.; Rabiser, R.; Salinesi, C. & Diaz, D (2011). Using Constraint Programming to Verify DOPLER Variability Models. *In 5th International Workshop on Variability Modelling of Software-intensive Systems (VaMos'11)*, pp.97-103, ACM Press. Belgium.

Mazo, R.; Salinesi, C.; Diaz, D. & Lora-Michiels, A. (2011). Transforming Attribute and Clone-Enabled Feature Models into Constraint Programs Over Finite Domains. *6th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, Springer Press, China.

Mazo, R.; Lopez-Herrejon, R.; Salinesi, C.; Diaz, D. & Egyed, A. (2011). A Constraint Programming Approach for Checking Conformance in Feature Models. *In 35th IEEE Annual International Computer Software and Applications Conference (COMPSAC'11)*, IEEE series, Germany.

Mazo, R.; Salinesi, C.; Djebbi, O.; Diaz, D. & Lora-Michiels, A. (2011). Constraints: the Heard of Domain and Application Engineering in the Product Lines Engineering Strategy. *International Journal of Information System Modeling and Design IJISMD* (accepted), to appear in November 2011.

Mazo, R.; Salinesi, C.; Diaz, D. (2011). Abstract Constraints: A General Framework for Solver-Independent Reasoning on Product Line Models. Accepted on INSIGHT - Journal of International Council on Systems Engineering (INCOSE), to be released the 15 October 2011.

Mendonça, M.; Wasowski, A. & Czarnecki, K. (2009). SAT–based analysis of feature models is easy. *In D. Muthig and J. D. McGregor, editors, SPLC, volume 446 of ACM International Conference Proceeding Series,* pp. 231-240. ACM.

Nuseibeh, B.; Kramer, J. & Finkelstein A. (1994)A framework for expressing the relationships between multiple views in requirements specification. *IEEE Trans. Software Eng*. 20(10) pp. 760–773.

Oxford University. 2008). *Concise Oxford English Dictionary*. Oxford University Press, UK.

Pohl, K.; Böckle, G.; van der Linden, F. (2005). Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, Heidelberg.

Riebisch, M.; Bollert, K.; Streitferdt, D.; Philippow, I. (2002). Extending feature diagrams with UML multiplicities, in: Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT2002), Pasadena, CA.

Salinesi, C.; Mazo, R. & Diaz, D. (2010). Criteria for the verification of feature models. *In Proceedings of the 28th INFORSID Conference*, pp. 293-308. France.

Salinesi, C.; Mazo, R.; Diaz, D. & Djebbi, O. (2010) Solving Integer Constraint in Reuse Based Requirements Engineering. *In 18th IEEE Int. Conference on Requirements Engineering (RE'10)* IEEE Computer Society pp. 243-251. Australia.

Salinesi, C. ; Mazo, R. ; Djebbi, O. ; Diaz, D. ; Lora-Michiels, A. (2011). Constraints: the Core of Product Line Engineering. Fifth IEEE International Conference on Research Challenges in Information Science (RCIS), IEEE Press, Guadeloupe-French West Indies, France.

Schobbens, P.Y.; Heymans, P.; Trigaux, J.C.; Bontemps Y. Generic semantics of feature diagrams, Journal of Computer Networks, Vol 51, Number 2 (2007).

Schulte, Ch.; Stuckey, P. J. (2008). Efficient constraint propagation engines. ACM Trans. Program. Lang. Syst., 31(1).

Segura, S. (2008). Automated Analysis of Feature Models using Atomic Sets. First Workshop on Analyses of Software Product Lines (ASPL'08), SPLC'08. Limerick, Ireland.

Stahl, T.; Völter, M. & Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management.* Wiley editors, San Francisco.

Streitferdt, D.; Riebisch, M.; Philippow, I. (2003). Details of formalized relations in feature models using OCL. In Proceedings of 10th IEEE International Conference on Engineering of Computer–Based Systems (ECBS 2003), Huntsville, USA. IEEE Computer Society, pages 45–54.

Trinidad, P.; Benavides, D.; Durán, A.; Ruiz-Cortés, A. & Toro, M. (2008). Automated error analysis for the agilization of feature modeling. J*ournal of Systems & Software*, 81(6) pp. 883-896, Elsevier.

Trinidad, P., Benavides, D., Ruiz-Cortés, A. (2006), A first step detecting inconsistencies in feature models. In CAiSE Short Paper Proceedings, Advanced Information Systems

Engineering, 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg.

Van Hentenryck, P. (1989). Constraint Satisfaction in Logic Programming. The MIT Press.

Van den Broek, P. & Galvão, I. (2009). Analysis of Feature Models using Generalised Feature Trees. *Third International Workshop on Variability Modelling of Software-intensive Systems VaMoS*. ICB-Research Report No. 29, Universität Duisburg-Essen, pp. 169-172. Spain.

Van der Storm, T. (2007). Generic Feature-Based Composition. In: M. Lumpe and W. Vandeperren, editors, Proceedings of the Workshop on Software Composition (SC'07), volume 4829 of LNCS, pp. 66-80, Springer.

Von der Maßen, T. ; Lichter, H. (2003). RequiLine: A requirements engineering tool for software product lines, Proceedings of International Workshop on Product Family Engineering PFE-5, Springer LNCS 3014, Siena, Italy.

Von der Maßen, T. & Lichter, H. (2004). Deficiencies in feature models. *In Tomi Mannisto and Jan Bosch, editors, Workshop on Software Variability Management for Product Derivation - Towards Tool Support*.

White, J.; Doughtery, B.; Schmidt, D. (2009). Selecting highly optimal architectural feature sets with filtered cartesian flattening. Journal of Systems and Software, 82(8):1268–1284.

Yan, H.; Zhang, W.; Zhao, H. & Mei, H. (2009). An optimization strategy to feature models' verification by eliminating verification-irrelevant features and constraints. *In the proceedings of the International Conference on Software Reuse (ICSR)*, pp. 65–75.

Zhang, W.; Zhao, H.; Mei, H. (2004) A Propositional Logic-Based Method for Verification of Feature Models. In: Proceedings of 6th International Conference on Formal Engineering Methods, pp. 115–130.

IntechOpen



Software Product Line - Advanced Topic

Edited by Dr Abdelrahman Elfaki

ISBN 978-953-51-0436-0
Hard cover, 122 pages
Publisher InTech
Published online 04, April, 2012
Published in print edition April, 2012

The Software Product Line (SPL) is an emerging methodology for developing software products. Currently, there are two hot issues in the SPL: modelling and the analysis of the SPL. Variability modelling techniques have been developed to assist engineers in dealing with the complications of variability management. The principal goal of modelling variability techniques is to configure a successful software product by managing variability in domain-engineering. In other words, a good method for modelling variability is a prerequisite for a successful SPL. On the other hand, analysis of the SPL aids the extraction of useful information from the SPL and provides a control and planning strategy mechanism for engineers or experts. In addition, the analysis of the SPL provides a clear view for users. Moreover, it ensures the accuracy of the SPL. This book presents new techniques for modelling and new methods for SPL analysis.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Camille Salinesi and Raúl Mazo (2012). Defects in Product Line Models and How to Identify Them, Software Product Line - Advanced Topic, Dr Abdelrahman Elfaki (Ed.), ISBN: 978-953-51-0436-0, InTech, Available from: http://www.intechopen.com/books/software-product-line-advanced-topic/defects-in-product-line-models-and-how-to-identify-them

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2012 The Author(s). Licensee IntechOpen. This is an open access article distributed under the terms of the Creative Commons Attribution 3.0 License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

IntechOpen

IntechOpen